AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data

Romain Egele, Prasanna Balaprakash, Venkatram Vishwanath, Isabelle Guyon, Zhengying Liu

AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data

Romain Egele
MSc&T AIViC
École polytechnique
Palaiseau, France
romain.egele@polytechnique.edu

Prasanna Balaprakash
Mathematics and Computer Science Division
Argonne Leadership Computing Facility
Argonne National Laboratory
Lemont, Illinois, USA
pbalapra@anl.gov

Venkatram Vishwanath
Argonne Leadership Computing Facility
Argonne National Laboratory
Lemont, Illinois, USA
venkat@anl.gov

Isabelle Guyon
INRIA-LRI, Université Paris-Saclay
Orsay, France
& *ChaLearn*, USA
guyon@chalearn.org

Zhengying Liu
TAU, INRIA-LRI-CNRS
Orsay, France
zhengying.liu@inria.fr

***Abstract*—Developing high-performing predictive models for large tabular data sets is a challenging task. The state-of-the-art methods are based on expert-developed model ensembles from different supervised learning methods. Recently, automated machine learning (AutoML) is emerging as a promising approach to automate predictive model development. Neural architecture search (NAS) is an AutoML approach that generates and evaluates multiple neural network architectures concurrently and improves the accuracy of the generated models iteratively. A key issue in NAS, particularly for large data sets, is the large computation time required to evaluate each generated architecture. While data-parallel training is a promising approach that can address this issue, its use within NAS is difficult. For different data sets, the data-parallel training settings such as the number of parallel processes, learning rate, and batch size need to be adapted to achieve high accuracy and reduction in training time. To that end, we have developed AgEBO-Tabular, an approach to combine aging evolution (AgE), a parallel NAS method that searches over neural architecture space, and an asynchronous Bayesian optimization method for tuning the hyperparameters of the data-parallel training simultaneously. We demonstrate the efficacy of the proposed method to generate high-performing neural network models for large tabular benchmark data sets. Furthermore, we demonstrate that the automatically discovered neural network models using our method outperform the state-of-the-art AutoML ensemble models in inference speed by two orders of magnitude while reaching similar accuracy values.**

***Index Terms*—AutoML, Deep Learning, Data Parallelism, Tabular, Big Data**

I. Introduction

Tabular data sets are often diverse. They are obtained from multiple sources and modes, where combining certain inputs using *problem-specific* domain knowledge typically leads to better and physically meaningful features and consequently robust models [1], [2]. Many high-performing predictive models for tabular data are based on classical supervised machine learning (ML) methods such as bagging, boosting, and kernel-based methods. Specifically, ensemble methods that combine models obtained from different supervised ML methods have emerged as state-of-the-art for a wide range of predictive modeling tasks with tabular data. However, design and development of such ensemble models is a highly iterative, manually intensive, and time-consuming task. Typically an ML pipeline for tabular data is composed of several components: data processing, dimension reduction, data balancing, feature selection, hyperparameter tuning, model selection, and ensemble strategy (such as stacking, bagging, and weighted combination). Given the design choices for each component, the complexity of designing an effective ML pipeline for tabular data is often beyond nonexperts.

Deep neural networks (DNNs) have achieved significant success in overcoming the issues of manual feature engineering and the complexities of developing classical supervised ML pipeline. Nevertheless, designing DNNs for tabular data has received relatively less attention compared with image and text data. From the methodological perspective, there are two main reasons. First, given the diversity of tabular data, designing DNNs with shared patterns such as convolutional and recurrent units is not meaningful unless further assumptions about the data are made. Second, fully connected DNNs, which are typically used for tabular data, can potentially lead to unsatisfactory performance because they can have large numbers of parameters, overfitting issues, and a difficult optimization landscape with low-performing local optima [3].

Automated machine learning (AutoML) is a promising approach to address the methodological challenges in developing DNNs for tabular data. Neural architecture search (NAS), a

class of AutoML, is an approach to automate development of customized DNNs for a given data set. The NAS methods can be grouped into individual search methods and weight-sharing methods. The former generate a large number of architectures from a user-defined search space, train and validate each of them, and use the accuracy values to improve the generated architectures. The main advantage of these methods is parallelization: the generated architectures are independent, and they can be trained simultaneously. The disadvantage is that since each architecture is trained from scratch, architecture evaluation is expensive and becomes a bottleneck for effectiveness. To alleviate this issue, researchers proposed a different approach where the trained weights or computations are shared from an architecture to another during the search. This is enabled by defining a search space as an overparameterized network [4] (also named hypernetwork), where the search samples subarchitectures and leverages the trained weights and computations from previously trained subarchitectures. This results in significant reduction of evaluation time for several tasks. Nevertheless, the disadvantage of these methods is the instability due to the optimization gap between the supernetwork and its subarchitectures. In particular, optimizing the hypernetwork does not necessarily results in high-quality subarchitectures [5].

We focus on individual NAS search for large tabular data because of its ability to leverage multiple compute nodes to find high-performing neural networks. Specifically, we adopt aging evolution (AgE) [6], a parallel NAS method that generates a population of neural architectures, training them concurrently using multiple nodes, and improves the population by performing mutations on the existing architectures within a population. To reduce the training time of each architecture, we utilize the widely used distributed data-parallel training technique. In this approach, the large training data is split into shards and distributed to multiple processing units. Multiple models with the same architecture are trained on different data shards, and the gradients from each model are averaged and used to update the weights of all the models. Combining an individual NAS search method with distributed data-parallel training is a challenging task because the combination of the two methods requires nested parallelism. Moreover, the distributed data parallelism requires data-set-specific tuning of parallelism, learning rate, and batch size in order to maintain accuracy and reduce training time. To that end, we make the following contributions:

- We develop AgEBO-Tabular, a joint neural architecture and hyperparameter search that combines aging evolution (AgE), a parallel NAS method [6] for searching the neural architecture space, and an asynchronous Bayesian optimization method for tuning the hyperparameters of data-parallel training. AgEBO-Tabular searches the architecture space and the hyperparameters of data-parallel training simultaneously.
- We evaluate the efficacy of the proposed approach on four large tabular data sets and show that AgE with the autotuned data-parallel training outperforms the accuracy of the AgE method by an order of magnitude less computation time.
- We demonstrate that an automatically discovered single neural network model is faster than the state-of-the-art automatically generated ensemble models with respect to inference speed by two orders of magnitude.

The novelty of our work is fourfold: developing a new method for joint neural architecture and hyperparameter search, accelerating NAS with data-parallel training, using asynchronous Bayesian optimization for tuning the hyperparameters of data-parallel training, and advancing the state-of-the-art in the design of DNNs for large tabular data.

II. Problem formulation

Let D_{train}, D_{valid}, and D_{test} are the training, validation, and test data, respectively. A neural architecture configuration h_a is a vector from the neural architecture search space H_a, defined by a set of neural architecture decision variables. A hyperparameter configuration h_m is a vector from hyperparameter search space H_m defined by a set of hyperparameters. The joint neural architecture and hyperparameter search space H is given by $H_a \times H_m$. The problem of joint neural architecture and hyperparameter search can be formulated as the following bilevel optimization problem:

$$\begin{aligned} h_a^*, h_m^* = & \underset{(h_a,h_m)\in H_a\times H_m}{\arg\max} \mathcal{M}_{w^*}^{val}(h_a,h_m) \\ \text{s.t. } w^* = & \underset{w}{\arg\min}\, \mathcal{L}_{h_a,h_m}^{train}(w), \end{aligned} \tag{1}$$

where $\mathcal{M}_{w^*}^{val}(h_a,h_m)$ is the validation accuracy that needs to be maximized on D_{valid} and $\mathcal{L}_{h_a,h_m}^{train}(w)$ is a loss function that needs to be minimised by optimizing the weights w of the neural network configured with (h_a,h_m) using D_{train}. The test data D_{test} is used only for the final evaluation.

The architecture search space differs from the hyperparameter search space with respect to the values that the decision variables take. All the decision variables in the architecture search space belong to categorical (nonordinal) type, where different values for a given variable do not have any particular order. On the other hand, the hyperparameter search space is characterized by mixed-integer variables. This comprises integer, real, binary, and categorical types. Often, the number of categorical hyperparameters is relatively smaller than that of other types. Note that when all the variables in hyperparameter search space belong to a categorical type, explicit partitioning in the search space is not required; consequently, a custom method such as our proposed AgEBO-Tabular for joint neural architecture and hyperparameter search becomes less relevant.

In our study, H_a is defined by the decision variables to construct fully connected neural networks with skip connections for tabular data, and H_m is defined by the hyperparameters of the data-parallel training (learning rate, batch size, and number of parallel processes).

III. AgEBO-Tabular

The AgEBO-Tabular approach that we propose comprises three components: neural architecture search space for tabular data, tunable data-parallel training as evaluation strategy, and the AgEBO algorithm for joint neural architecture and hyperparameter search.

A. Neural architecture search space for tabular data

We model the search space of the neural architecture using a directed acyclic graph, which starts and ends with input and output nodes, respectively. They are fixed based on the input and output dimensions of the tabular data, respectively. Between these two nodes are intermediate nodes, each of which can be a variable or a skip-connection node. Each node represents a categorical decision variable that can take a list of nonordinal values. Each variable node represents a dense layer with a list of different layer types; the choice is made by the NAS method. The skip connections between the variable nodes are created by using skip-connection nodes. This type of node has two choices: zero for no skip connection and identity for the creation of skip connection. Given a pair of consecutive variable nodes $\mathcal{N}_k$, $\mathcal{N}_{k+1}$, three skip-connection nodes $\mathcal{SC}^{k+1}_{k-3}, \mathcal{SC}^{k+1}_{k-2}, \mathcal{SC}^{k+1}_{k-1}$ are created. The choice of identity for these skip-connection nodes respectively allows for connection to the three previous nonconsecutive variable nodes $\mathcal{N}_{k-3}, \mathcal{N}_{k-2}, \mathcal{N}_{k-1}$. For example, if identity is chosen for $\mathcal{SC}^{k+1}_{k-1}$, a skip connection is made between $\mathcal{N}_{k-1}$ and $\mathcal{N}_{k+1}$ by passing the tensor output from $\mathcal{N}_{k-1}$ through a linear layer and a sum operator. The linear layer is used to project the tensor from $\mathcal{N}_{k-1}$ to a correct shape. This is required for the creation of skip connections between $\mathcal{N}_{k-1}$ and $\mathcal{N}_{k+1}$ when their number of neuron units is different. The sum operator adds the projected input tensor from $\mathcal{N}_{k-1}$ and the tensor from $\mathcal{N}_k$, passes the summed tensor through the $ReLu$ activation function, and sends the resulting tensor as input to $\mathcal{N}_{k+1}$. When $\mathcal{SC}^{k+1}_{k-2}$ and $\mathcal{SC}^{k+1}_{k-3}$ take identity values, the tensors from $\mathcal{N}_{k-2}$ and $\mathcal{N}_{k-3}$ undergo the same linear projection, and the tensor is given to the sum operator. When there is no skip connection, $\mathcal{SC}^{k+1}_{k-3}, \mathcal{SC}^{k+1}_{k-2}, \mathcal{SC}^{k+1}_{k-1}$ are set to zero; $\mathcal{N}_k$ and $\mathcal{N}_{k+1}$ are fully connected without the linear layer and the sum operator. The same process is repeated for each of the m variable nodes. See Figure 1 for an example.

The dense layer type is defined by the number of units and the activation function. For the former and the latter we used {16, 32, 48, 64, 80, 96} and {Identity, Swish [7], ReLu, Tanh, Sigmoid}. These resulted in 31 (6 units $\times$ 5 activation functions, and an identity) dense layer types for each variable node. Although one can order the 31 values using the number of units in the layer, we did not consider and leverage such order from the generality perspective. For example, if we consider only one value for the unit and different activation functions, then we cannot order the values in the list and cannot leverage the ordering in the NAS search. We set the maximum number of variable nodes to 10. Consequently, we have 37 decision variables composed of 10 variable nodes and 27 skip-connection nodes. The first variable node will not have a skip connection node. The second and the third variable nodes have 1 and 2 skip-connection nodes, respectively. The fourth to tenth variable nodes have 3 skip-connection nodes each. The output node has 3 skip connections as well. Consequently, the total number of architectures in the search space is $31^{10} \times 2^{27} \approx 1.1 \times 10^{23}$.

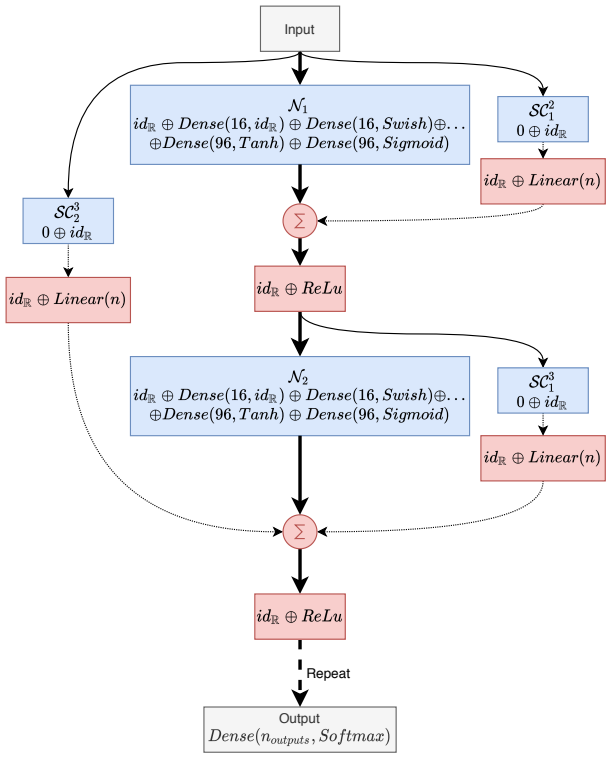


Fig. 1: Neural architecture search space. The nodes $\mathcal{N}_1$ and $\mathcal{N}_2$ represent dense layers $Dense(x, y)$, where x is the number of neurons and y is the activation function. The nodes $\mathcal{SC}^2_1, \mathcal{SC}^3_1, \mathcal{SC}^3_2$ represent the possible skip-connection nodes, when $id_{\mathbb{R}}$ is chosen for each of them. The node $\mathcal{N}_2$ is connected to input node through $\mathcal{SC}^2_1$. The output node is connected to input and $\mathcal{N}_1$ nodes through $\mathcal{SC}^3_1$ and $\mathcal{SC}^3_2$, respectively. The nodes shown in red are used to manage the different tensor sizes and apply an element-wise sum (represented by the cross inside a circle).

B. Tunable data-parallel training as evaluation strategy

The evaluation of an architecture in the individual NAS method consists of training the network and computing the validation accuracy. To speed up the evaluation, we use distributed data-parallel training. Given a neural architecture $\mathcal{A}$, the training data set is split in n mutually exclusive subsets called shards, which are given to n parallel processes. Each of the n processes trains a copy of the same neural architecture $\mathcal{A}$ on its own shard. The gradients from each copy of neural architecture are synchronized and are used to update the weights. Moreover, we use the widely used linear scaling rule [8] to adapt the learning rate and batch size depending on the level of parallelism in the data-parallel training. This heuristic

states that the learning rate lr_n and batch size bs_n with n processes should be scaled linearly with respect to n:

$$lr_n = n * lr_1; bs_n = n * bs_1, \tag{2}$$

where lr_1, bs_1 are respectively the learning rate and batch size used for training with a single process. We treat n, lr_1, and bs_1 as hyperparameters and tune them using Bayesian optimization. By leveraging the linear scaling rule, we try to achieve linear scaling for training time; however, there is an upper linear scaling limit above which the accuracy will suffer (without advanced and sophisticated layer-wise learning rate and adaptive batch size). Therefore, by tuning n, lr_1, and bs_1, we try to find the upper linear scaling limit that gives maximal reduction in training time without losing accuracy.

C. AgEBO: Aging evolution with Bayesian optimization

To perform joint a neural architecture and hyperparameter search, we propose aging evolution with Bayesian optimization (AgEBO). Our method combines AgE, a parallel NAS method, for searching over the architecture space, and asynchronous Bayesian optimization (BO), for tuning the hyperparameters data-parallel training.

Algorithm 1 shows the pseudo code of AgEBO. The method follows the manager-worker paradigm for parallelization. It starts with W workers, each with a maximum of n_{max} parallel processing units for data-parallel training. The initialization phase starts by allocating an empty queue for the population of size P and BO optimizer object. It is followed by sampling W architecture configurations and hyperparameter configurations, respectively, and concatenating them. The neural network models are built by using the resulting configurations and are sent for concurrent evaluation on W workers by using the submit_evaluation interface (lines 3–7). Each worker uses the learning rate, batch size, and number of processes from the configuration that it received to run the data-parallel training. The iterative part of the algorithm consists of collecting the results (validation accuracy values) once workers finish their evaluation (line 9) and using them for generating the next set of architecture and hyperparameter configurations for evaluation. The BO optimizer object takes the hyperparameter configurations and their corresponding validation accuracy values and generates a $|results|$ number of hyperparameter values (using optimizer.tell and optimizer.ask interfaces, respectively, lines 12–13). To generate $|results|$ number of architecture configurations, the following steps are performed repeatedly: random sampling S architecture configurations from the incumbent population, selecting the best, and applying a random mutation to generate a child model hyperparameter configuration (lines 16–18). The generated architecture and hyperparameter configurations are concatenated and sent for evaluation. Note that in the beginning of the search, the population queue does not have P number of finished evaluations (given that all evaluations do not necessarily finish in the same time). Therefore, the architecture configurations are generated at random while the population size is smaller than P (line 20) The mutation corresponds to choosing a different operation for one variable node in the search space. This is achieved by first randomly selecting a variable node and then choosing (again at random) a value for that node excluding the current value. Then, the child is added to the population by replacing the oldest member of the population.

Algorithm 1: AgE (black) and AgEBO (black + blue)

```
   inputs: P: population size, S: sample size, W: workers
   output: highest-accuracy model in history
   /* Initialization                                    */
1  population ← create_queue(P) // Alloc empty Q of
       size P
2  optimizer ← optimizer()
3  for i ← 1 to W do
4  |  model.h_a ← random_point(H_a)
5  |  model.h_m ← random_point(H_m)
6  |  submit_evaluation(model) // Nonblocking
7  end
   /* Main loop                                         */
8  while not done do
   |  // Query results
9  |  results ← get_finished_evaluations ()
10 |  if |results| > 0 then
11 |  |  population.push(results) // Aging population
   |  |  // Generate hyperparameter configs
12 |  |  optimizer.tell(results.h_m, results.valid_accuracy)
13 |  |  next ← optimizer.ask(|results|)
   |  |  // Generate architecture configs
14 |  |  for i ← 1 to |results| do
15 |  |  |  if |population| = P then
16 |  |  |  |  sample ← random_sample(population,S)
17 |  |  |  |  parent ← select_parent(sample)
18 |  |  |  |  child.h_a ← mutate(parent.h_a)
19 |  |  |  else
20 |  |  |  |  child.h_a ← random_point(H_a)
21 |  |  |  end
22 |  |  |  child.h_m ← next[i].h_m
23 |  |  |  submit_evaluation(child) // Nonblocking
24 |  |  end
25 |  end
26 end
```

The BO component of AgEBO optimizes the hyperparameters (h_m) by marginalizing the architecture decision variables (h_a). The BO method generates hyperparameter configurations as follows. It starts by sampling a large number of unevaluated hyperparameter configurations. For each sampled configuration h_m^i, it uses a model M to predict a point estimate (mean value) $\mu(h_m^i)$ and standard deviation $\sigma(h_m^i)$. The sampled hyperparameter configurations are ranked by using the upper-confidence bound (UCB) acquisition function:

$$UCB(h_m^i) = \mu(h_m^i) + \kappa\sigma(h_m^i), \tag{3}$$

where $\kappa \geq 0$ is a parameter that controls the trade-off between exploration and exploitation. A value of $\kappa = 0$ corresponds to pure exploitation, where the hyperparameter configuration with the lowest mean value is always selected. A large value of κ corresponds to exploration, where hyperparameter configurations with large variance are selected. Evaluation of such configurations results in improvement of the model M. A typical BO optimization method with UCB is sequential

and generates only one hyperparameter configuration at a time. This is not useful in our setting given the scale required by the AgE method. Therefore, to generate multiple hyperparameter configurations at the same time, we adopt an asynchronous BO that leverages multipoint acquisition function based on a constant liar strategy. This approach starts by selecting a hyperparameter that maximizes the UCB function. The model M is retrained with the selected hyperparameter configuration and a dummy value (lie). The next hyperparameter configuration is obtained by maximizing the UCB function using the updated model. The process of selecting a configuration and retraining the model with a lie is repeated until a required number of configurations are sampled. The mean of all the validation accuracy values found up to that point is used as a lie. While several sophisticated asynchronous BO methods exist, the adoption of the constant liar strategy is motivated by its computational simplicity and low overhead. Since the mutation operation in AgE method is simple, the BO method needs to generate multiple configurations in short computation time. Failure to do so will adversely affect the overall node utilization.

D. Implementation details

We implemented AgEBO in DeepHyper [9], open-source scalable AutoML software designed for neural architecture and hyperparameter search. A high-level implementation overview of the AgEBO method is shown in Figure 2. Algorithm 1 runs on a single process $\mathcal{P}$. DeepHyper leverages the Balsam workflow system [10] to schedule the evaluation of architectures concurrently. Specifically, the submit_evaluation interface of AgEBO calls the Balsam workflow system, which is responsible for running the architecture training on W workers (via mpirun), collecting the validation accuracy values, and returning the results through a get_finished_evaluations interface. We allocate one compute node for the search. For the BO module implementation, we used scikit-optimize package [11] and its ask and tell interface. The random forest method is used as the model M within BO. We use the Horovod library [12] for the distributed data-parallel training implementation within AgEBO. The AgEBO-Tabular code is open-sourced and accessible on the DeepHyper Github repo.[1]

IV. Experiments

We used four large tabular data sets from the OpenML [13] benchmark. The selection was motivated by a tabular data benchmark study using AutoGluon [1], a recently proposed state-of-the-art AutoML method for tabular data. Among all the data sets benchmarked with AutoGluon, we selected the following four largest data sets having the largest number of data points:

1) Covertype [14]: It contains 581,012 data points, 54 input features, and 7 classes. The task is to predict the forest cover type given cartographic variable input data.

[1]https://github.com/deephyper/NASBigData

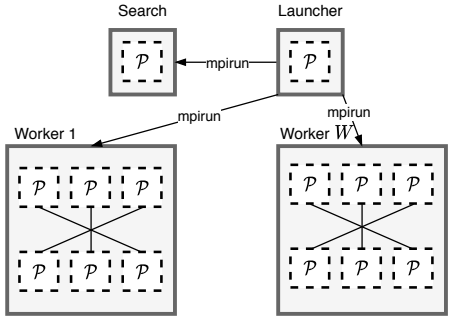


Fig. 2: Overview of AgEBO implementation. The AgEBO search runs on a single process and uses the Balsam workflow system to run the architecture evaluation on W workers using the mpirun interface.

2) Airlines [15]: It contains 539,383 data points, 8 input features, and 2 classes. The task is to develop a model to indicate whether a given flight will be delayed or not given input data of the scheduled departure.
3) Albert [16]: It contains 425,240 data points, 79 input features, and 2 classes from the AutoML Challenge series (2015–2018).
4) Dionis [16]: It contains 416,188 data points, 61 input features, and 355 classes from the AutoML Challenge series (2015–2018).

For each data set, we grouped the data for training, validation, and testing as in the Auto-PyTorch benchmark study. Specifically, we used 42% for training, 25% for validation, and 33% for testing. In all the AutoML methods, we used the training and validation data set within AgEBO-Tablular. The selected best model was evaluated on the testing data.

Experiments were run on the Theta supercomputer at the Argonne Leadership Computing Facility (ALCF). Theta is a Cray XC40 11.69-petaflops system composed of 4,392 nodes with Intel Knights Landing CPUs of 64 cores each equipped of 192 GB of DDR4 memory. Since the data set that we consider fits in a single-node memory, we did not utilize multinode data-parallel training. Instead, the data-parallel training within AgEBO was limited to single node; however, it uses multiple processes within the single node to accelerate training. The number of threads per process within the single node, tpr, is set to the ratio of the number of threads per node, tpn, and the number of process per node, rpn. The threading is configured based on guidelines provided by the ALCF, which is based on TensorFlow documentation: intrathreads = `OMP_NUM_THREADS` = tpr; interthreads = 2; CPU affinity = depth (equivalent to: `KMP_AFFINITY` = "granularity=fine,verbose,compact,1,0"); `KMP_BLOCK_TIME` = 0.

By default, the NAS experiments were run for a wall time of 3 hours on 129 nodes of Theta. One node was reserved for the search, and 128 nodes were used as workers to train and validate the models within AgEBO.

AgE was used as the baseline. The optimizer was set to

Adam [17], and each model was evaluated for 20 epochs of training. A gradual warmup strategy [18] was employed for the first 5 epochs. A callback was used to automatically reduce the learning rate on a plateau with a patience of 5 epochs. The objective in the AutoML methods is to maximise the the validation accuracy. For the search, the population (P) and sample sizes (S) were set to 100 and 10, respectively. The batch size and learning rate were set to 256 and 0.01, respectively. AgEBO variants adopt the same training strategy as AgE uses. The difference between AgEBO variants and AgE is that the values of the batch size, learning rate, and number of processes for data-parallel training can be tuned concurrently along with the architecture search.

The range for hyperparameters of data-parallel training was set as follows: batch size (bs_1) $\in$ [32, 64, 128, 256, 512, 1024]; learning rate (lr_1) $\in$ (0.001, 0.1), which are sampled in a log-uniform scale within BO; and number of processes (n) $\in$ [1,2,4,8].

A. Impact of static data-parallel training on AgE

We show that the accuracy of the architectures discovered by the AgE method with data-parallel training deteriorates significantly without tuning the learning rate, batch size, and number of processes.

We evaluated AgE with data-parallel training without BO but varied the number of processes. We used the default learning rate and batch size for $n = 1$. The learning rate and batch size for different numbers of processes were scaled by using the linear scaling rule. We ran the experiments on the Covertype data set.

The results are shown in Figure 3 and Table I, where AgE-n refers to AgE with n processes for data-parallel training. From the results, we observe that increasing the number of ranks from 1 to 4 per evaluation increases the accuracy. This increase can be attributed to the reduced training time for architecture evaluation, which increases the number of evaluated architectures from 632 to 2,421. Nevertheless, for AgE-8, we observe that the accuracy significantly decreases despite the large number (4,221) of evaluated architectures. The poor accuracy of AgE-8 can be attributed to the scaled learning rate and batch size values for 8 processes and/or the possibility that 8 is not right value for achieving reduction in training time without losing accuracy.

	AgE-1	AgE-2	AgE-4	AgE-8
Number of architectures	632	1764	2421	4221
Training time (min.)	26.54 ± 7.68	8.97 ± 0.76	5.38 ± 0.4	3.19 ± 0.29
Validation accuracy	0.918	0.925	0.925	0.902

TABLE I: Results for static data-parallel training in AgE.

B. Impact of autotuned data-parallel training within AgEBO

Here, we show that tuning the learning rate, batch size, and number of processes through BO improve both the accuracy and time to solution.

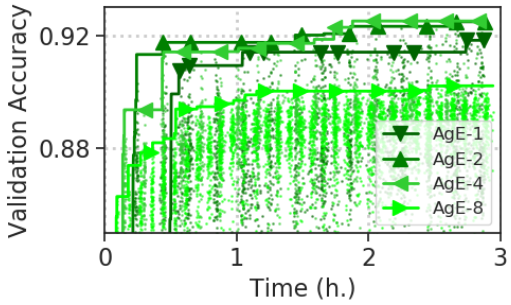


Fig. 3: Search trajectory of AgE with different numbers of processes for data-parallel training on Covertype data set. The thick lines denote the best validation accuracy over time for each method so far. The dots denote the validation accuracy of each architecture found during the search.

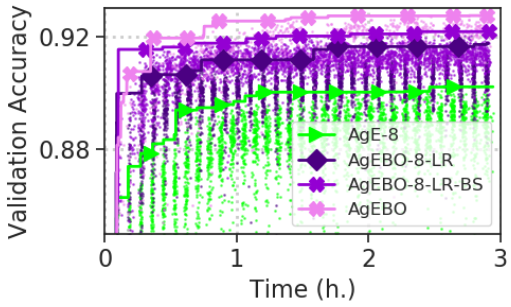


Fig. 4: Search trajectory of AgEBO variants and AgE-8 on Covertype data set. See Fig. 3 caption for the notations used (LR – learning rate, BS – batch size).

To analyze the effectiveness of BO within AgEBO, we compared it with two of its variants. AgEBO-8-LR and AgEBO-8-LR-BS. In the former, only the learning rate was tuned by setting the batch size and the number of processes for the data-parallel training to the default batch size and 8, respectively. In the latter, the batch size and learning rate were tuned by setting the number of processes to 8. As a baseline, we used AgE-8. The experiments were run on the Covertype data set.

The results are shown in Figure 4. We can observe that the AgEBO variants outperform AgE-8 with respect to both accuracy and the time to reach that accuracy. The comparison between AgEBO-8-LR and AgE-8 shows that tuning the values of the learning rate leads to significant improvement with respect to both accuracy and time to solution. Similarly, AgEBO-8-LR-BS achieves a higher accuracy value than that of AgEBO-8-LR within a shorter time. However, AgEBO, which tunes all three hyperparameters, outperforms AgEBO-8-LR-BS. An exception is in the initial phases of the search (first 30 minutes), which is due to the initial rank exploration of AgEBO and its impact on the training time. Specifically, this can be attributed to the exploration of different parallelism settings during that phase, which increases the evaluation time of the architectures.

To ensure that the observed superior accuracy of AgEBO is not by chance, we analyzed the number of unique architectures found over time that have a validation accuracy higher than 0.90 for AgE-n variants and AgEBO. The threshold of 0.90 is computed by taking the minimum of 0.99-quantiles of validation accuracy for each variant. The results are shown in Figure 5. We observe that AgEBO obtains a larger number of high-performing architectures than that of AgE-n variants. Moreover, despite given the same number of nodes, AgEBO is twice as fast as AgE-n variants in reaching the same number of high-performing architectures. Specifically, AgE-4 and AgE-8 obtain 10^2 high-performing architectures in 180 minutes whereas AgEBO obtains the same number within 90 minutes.

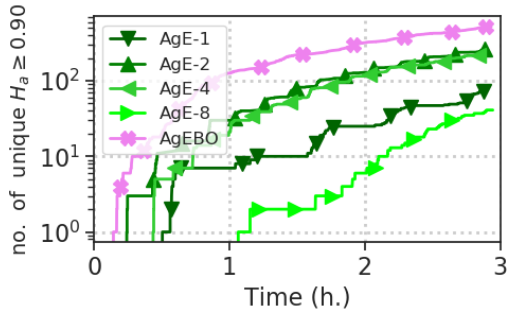


Fig. 5: Number of unique high-performing models obtained by AgEBO and AgE-n variants on the Covertype data set.

C. Comparison with AutoGluon and Auto-PyTorch

Here, we show that the prediction accuracy of our method is better than or comparable to that of the two state-of-the-art AutoML software AutoGluon [1] and Auto-PyTorch [2] while reducing the inference time of final models.

The two methods rely on ensemble approaches to boost their prediction accuracy values. AutoGluon combines different supervised learning models such as neural networks, LightGBM, CatBoost, random forest, extra trees, and K-nearest neighbors, the hyperparameters of which are automatically tuned. On the other hand, Auto-PyTorch adopts only neural network models but uses an ensemble strategy to improve the accuracy. We compared AgEBO with AutoGluon and Auto-PyTorch on all four data sets. We used AgE-1 as a baseline.

AutoGluon was run on a single node with a time limit of 4 hours for the call to the fit method to compensate for possible issues with the time estimation performed by the software. The `hyperparameter_tune=True` and `auto_stack=True` were set to maximise the accuracy as much as possible. The test accuracy was computed separately by reloading saved models. Table II shows the accuracy values of the best models and the corresponding inference time of AgEBO and AutoGluon. We observe that the test accuracy values of AgEBO and AutoGluon are comparable on all four data sets. However, the key advantage stems from the inference time with the trained model. Given that AgEBO generates a single neural network model, the inference time is between 2.7 and 4.3 seconds. On the other hand, AutoGluon relies on stacking a number of models, resulting in an inference time of about 7 minutes.

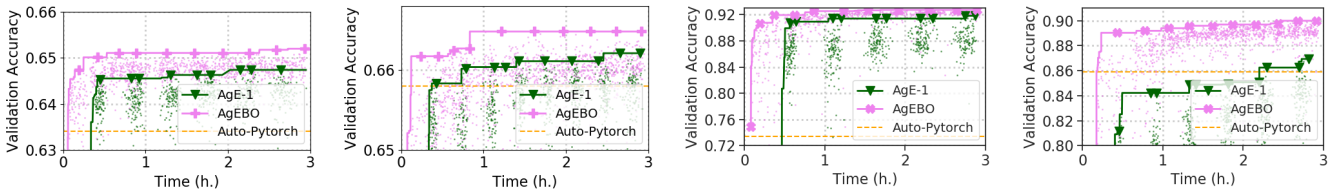
For Auto-PyTorch, since we cannot install the software in our ALCF Theta software stack because of software dependency issues, we used the results from the LCBench data base [19], which stores the results of experimental runs of the four data sets. We note, however, that although we used the same proportion of the training, validation, and testing split, the exact data splits were not used, the details of which are not available. Moreover, we did not compare against test accuracy from the ensemble strategy from Auto-PyTorch because we cannot retrieve ensemble strategy results from the LCBench data base. Therefore, we focus on comparison with validation accuracy values. Figure 6 shows the comparison between the best validation accuracy values found by AgEBO and Auto-PyTorch. We can observe that AgEBO achieves validation accuracy values that are higher than those of Auto-PyTorch within 30 minutes of search time. The observed differences in the accuracy values can be explained by two factors. First, Auto-PyTorch is not designed to generate a single neural network model but to generate multiple models and combine them using an ensemble strategy to have a good accuracy. Second, the architecture space of Auto-PyTorch is restricted to a smaller number of trainable parameters and smaller number of layers.

The comparison between AgE-1 and AgEBO in Figure 6 summarizes the benefits of autotuned data-parallel training. For the Airlines data set, the maximal accuracy found with AgE-1 is 0.647 at 121 minutes, whereas AgEBO finds a greater accuracy after 14 minutes and reaches its maximal accuracy of 0.652 after 163 minutes. For the Albert data set, the maximal accuracy found with AgE-1 is 0.662 at 147 minutes, wheress AgEBO achieves a higher accuracy after 36 minutes and reaches its maximal accuracy of 0.665 after 49 minutes. For Covertype, the maximal accuracy found with AgE-1 is 0.918 at 164 minutes, whereas AgEBO achieves a greater accuracy after 20 minutes and reaches its maximal accuracy of 0.927 after 165 minutes. For the Dionis data set, the maximal accuracy found with AgE-1 is 0.869 at 163 minutes, whereas AgEBO achieves a greater accuracy after 11 minutes and reaches its maximal accuracy of 0.900 after 147 minutes. In summary, AgEBO outperforms the AgE-1 with respect to both accuracy values and time to reach those accuracy values.

	AgEBO		AutoGluon	
data set	**Test Accuracy**	**Inference Time (s)**	**Test Accuracy**	**Inference Time (s)**
Airlines	**0.652 ± 0.002**	3.1	0.641	1124.9
Albert	0.661 ± 0.001	2.7	**0.688**	409.3
Covertype	**0.963 ± 0.001**	4.3	0.961	906.6
Dionis	**0.915 ± 0.0005**	3.2	0.907	1900.5

TABLE II: Test accuracy values and inference times obtained by AgEBO and AutoGluon on the four data sets.

Across all the four data sets, we observed that the node



(a) Airlines (b) Albert (c) Covertype (d) Dionis

Fig. 6: Search trajectory of AgE-1, AgEBO, and Auto-Pytorch on the four data sets. A horizontal dotted line shows the validation accuracy at the 20$^{\text{th}}$ epoch of the model with the best validation accuracy found by Auto-PyTorch. See Fig. 3 caption for the notations used.

utilization of AgEBO is similar to that of AgE—both reach an average value of ≈94%. This can be attributed to the effectiveness of the asynchronous BO that generates hyperparameter configurations with minimal overhead, which are combined with architecture decision variable values and sent for evaluation with minimal delay.

Table III shows the best hyperparameters obtained by AgEBO for the top 5 best-performing models on the four data sets. Note that AgEBO finds different hyperparameter configurations for different data sets to accelerate data-parallel training. Within the same data set, the hyperparameter configurations obtained for the best models are similar. These results demonstrate the need for data-set-specific hyperparameter tuning for data-parallel training, which is enabled by AgEBO.

We visualized the top 1% configurations based on the validation accuracy values obtained on all four data sets using principal component analysis. This is done by projecting the 37 architecture decisions and 3 hyperparameters of the top 1% configurations into two dimensions, respectively. The results are shown in Figure 7. From the results we can see a similar pattern. Each data set requires different values for architecture decision variables and data-parallel training hyperparameters.

	batch size	learning rate	no. of processes	validation accuracy
Airlines	64.0	0.001474	2.0	0.652008
	64.0	0.001250	2.0	0.651774
	128.0	0.001541	2.0	0.651086
	128.0	0.001742	2.0	0.651086
	64.0	0.001538	2.0	0.65090
Albert	128.0	0.005726	4.0	0.664827
	64.0	0.002226	2.0	0.664808
	64.0	0.002304	2.0	0.664552
	64.0	0.002490	2.0	0.664446
	64.0	0.002154	2.0	0.664190
Covertype	256.0	0.001392	1.0	0.927418
	256.0	0.001371	1.0	0.927325
	256.0	0.001409	1.0	0.927317
	256.0	0.001394	1.0	0.927309
	256.0	0.001394	1.0	0.927294
Dionis	256.0	0.001201	4.0	0.899902
	256.0	0.001237	4.0	0.899192
	256.0	0.001211	4.0	0.898837
	256.0	0.001159	4.0	0.898482
	256.0	0.001159	4.0	0.898260

TABLE III: Data-parallel training hyperparameter values obtained by AgEBO for the top 5 best models on the four data sets.

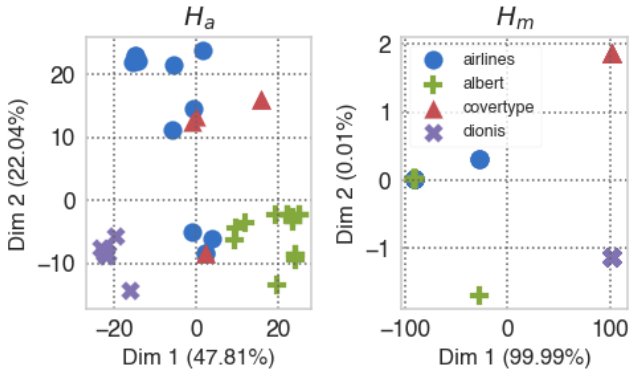


Fig. 7: Principal component analysis projection of top 1% configurations of architecture decision variables (H_a) and data-parallel training hyperparameters (H_m). The % on each axis shows the conserved variance (more than 80%) in two-dimensional projections.

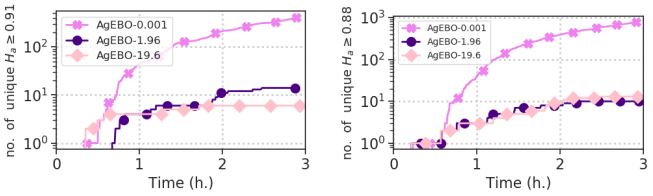
D. Exploration and exploitation in AgEBO

Here, we study the effect of exploration and exploitation of BO within AgEBO by varying κ values. We show that stronger exploitation is critical for the effectiveness of AgEBO.

The κ value in Eq. 3 controls the trade-off between exploration and exploitation in BO. In addition to the default κ value of 0.001, we ran AgEBO with two values: $\{1.96, 19.6\}$. Note that 1.96 is the typical κ value in Scikit-Optimize, which provides a balance between exploration and exploitation. The value of 19.6 is selected to enforce large exploration. We ran the experiments on the Covertype and the Dionis data sets.

Figure 8 shows the number of high-performing architectures found by AgEBO for three different κ values. The threshold was computed by computing 99% quantiles of the validation accuracy values for the three variants and taking the smallest value. We can observe that for both data sets, AgEBO with the default κ value of 0.001 (stronger exploitation) completely outperforms those with 1.96 (balance between exploration and exploitation) and 19.6 (stronger exploration) with respect to the number of high-performing architectures (between one and two orders of magnitude) and time needed to reach the number of the other two variants (between 2x and 3x faster).

The exploration of hyperparameter values in AgEBO with κ value of 0.001 happens only in the random initialization phase. During the iterative phase, given the stronger exploitation setting, hyperparameter configurations are generated close the best ones found so far in the search. On the other hand, there is a significant degree of exploration with κ values of 1.96 and 19.6. This results in increased data-parallel training time, which eventually reduces the generation of number of high-performing architectures.



(a) Covertype (b) Dionis

Fig. 8: Number of unique high-performing architectures discovered by AgEBO over time with different κ values.

V. Related Work

From the novelty perspective, our method has four components: hyperparameter search for data-parallel training, acceleration of NAS with data-parallel training, joint NAS and HPS, and application to tabular data. We review the related work from the perspective of each component and highlight our contributions.

The literature on HPS for tuning the hyperparameters on distributed data-parallel training to optimize learning rate, batch size, and number of processes is limited. A commonly used approach to adapt learning rate and batch size in distributed data-parallel training is the linear scaling rule. The values of the learning rate and batch size used for the single-process training are multiplied by the number of processes in distributed data-parallel training. In an Amazon blog [20], the importance of tuning learning rate and batch size for a given number of GPUs in data-parallel training has been discussed. Specifically, the Amazon SageMaker HPO tool has been used as a proof of concept; but the study was not performed at scale, and the effectiveness was not assessed on wide rage of data sets. The use of BO to tune the learning rate, batch size, and number of parallel processes in distributed training has never been investigated in the literature.

Within NAS, several approaches have been proposed to reduce the training time. Examples include using smaller architectures for the search and stacking them at the last step [21], [22], reducing the number of epochs [23], computing the validation performance from a randomly initialised DNN [24], estimating the accuracy performance of DNN for a large budget (time) when trained with a smaller budget [25], sharing the weights of previously trained DNN [4], imposing a time budget [26], and using information from data relatively to an initialised DNN (but only for convolution NN) without training [27]. These methods have several limitations. Stacking the simpler model is feasible for image data sets but can lead to overfitting in tabular data sets; and reducing the epochs and time budget during NAS can lead to poor relative ranking between the small and extensive budget and eventually result in low performing model [23]. Compared with all these methods, distributed data-parallel training is a generic and promising approach because of its ability to match with the learning curve of the classical training while consequently speeding up the training [18]. Nevertheless, the use of data-parallel training within NAS was not investigated in the literature.

The joint NAS and HPS approach that we propose is similar to BO Hyperband (BOHB) [23]. It considers the joint space and uses a multivariate kernel density estimation model to sample promising configurations. The sampled configurations are evaluated by using a successive halving approach, where promising configurations are allowed to run longer with more resources. Our approach differs from BOHB in the following ways. BOHB does not differentiate the model hyperparameters from algorithmic hyperparameters. It does not utilize data-parallel training to speedup the search, instead adopt successive halving. This is a blocking approach. Although quite effective under limited compute resource setting, scaling the successive halving method can lead to poor node utilization.

AutoML for tabular data has received considerable attention in recent years. Notable examples include auto-sklearn [28], Auto-WEKA [29], H2O AutoML [30], and TPOPT [31]. A benchmark [32] of these methods was conducted to compare their performance on different data sets. The auto-sklearn approach proved more robust in general. Recently, AutoGluon [1] and Auto-PyTorch [2] have emerged as state-of-the-art AutoML methods for tabular data. AutoGluon uses an ensemble of many different learning algorithms to then boost their performance. Auto-PyTorch also uses an ensemble approach, but models are restricted to DNNs. We showed that the prediction accuracy of AgEBO is better than or comparable to that of AutoGluon and Auto-PyTorch and provides a significant advantage with respect to the inference time.

VI. Conclusion and Future Work

We developed AgEBO-Tabular, a joint neural architecture and hyperparameter search method to discover high-performing neural network models for tabular data. We developed an architecture search space for generating fully connected neural networks with skip connections. The search method combines two distinct methods: (1) aging evolution (AgE), a parallel neural architecture search method to search over the architecture decision variables; and (2) an asynchronous Bayesian optimization (BO) method to automatically tune the hyperparameters of data-parallel training in order to reduce evaluation time of each architecture.

We showed that using data-parallel training in AgE without tuning the learning rate, batch size, and number of processes can affect the accuracy. Then, we demonstrated that AgEBO can improve the accuracy of the discovered models and the time to generate high-performing neural networks. We compared the best-discovered models from AgEBO with

AutoGluon and Auto-Pytorch, two state-of-the-art AutoML methods for tabular data, and showed the efficacy with respect to inference time and accuracy. The analysis of the best values obtained by AgEBO showed the need for data-set-specific tuning. Moreover, we showed that, unlike typical BO that balances the exploration and exploitation, a stronger exploitation is critical for AgEBO for generating high-performing models in short computation time.

Our future work will include (1) applying AgEBO to generate neural architectures for other data types such as image, texts, and graphs; (2) developing multinode data-parallel training within NAS for large data sets; and (3) developing meta-learning and transfer learning approaches to reuse the knowledge and results from previous experimental runs for related data sets.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility.

REFERENCES

[1] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data," *arXiv:2003.06505 [cs, stat]*, Mar. 2020, arXiv: 2003.06505. [Online]. Available: http://arxiv.org/abs/2003.06505

[2] L. Zimmer, M. Lindauer, and F. Hutter, "Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL," *arXiv:2006.13799 [cs, stat]*, Jun. 2020, arXiv: 2006.13799. [Online]. Available: http://arxiv.org/abs/2006.13799

[3] M. Fernandez-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?" p. 49.

[4] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient Neural Architecture Search via Parameter Sharing," *arXiv:1802.03268 [cs, stat]*, Feb. 2018, arXiv: 1802.03268. [Online]. Available: http://arxiv.org/abs/1802.03268

[5] X. Chu, T. Zhou, B. Zhang, and J. Li, "Fair DARTS: Eliminating Unfair Advantages in Differentiable Architecture Search," *arXiv:1911.12126 [cs, stat]*, Mar. 2020, arXiv: 1911.12126. [Online]. Available: http://arxiv.org/abs/1911.12126

[6] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized Evolution for Image Classifier Architecture Search," *arXiv:1802.01548 [cs]*, Feb. 2018, arXiv: 1802.01548. [Online]. Available: http://arxiv.org/abs/1802.01548

[7] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2018. [Online]. Available: https://openreview.net/forum?id=SkBYYyZRZ

[8] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: training imagenet in 1 hour," *CoRR*, vol. abs/1706.02677, 2017. [Online]. Available: http://arxiv.org/abs/1706.02677

[9] P. Balaprakash, R. Egele, M. Salim, V. Vishwanath, S. Wild, D. Jha, M. Dorier, K. G. Felker, R. Maulik, and B. Lusch, "deephyper/deephyper: 0.1.12," Oct. 2020. [Online]. Available: https://github.com/deephyper/deephyper

[10] M. A. Salim, T. D. Uram, J. T. Childers, P. Balaprakash, V. Vishwanath, and M. E. Papka, "Balsam: Automated Scheduling and Execution of Dynamic, Data-Intensive HPC Workflows," *arXiv:1909.08704 [cs]*, Sep. 2019, arXiv: 1909.08704. [Online]. Available: http://arxiv.org/abs/1909.08704

[11] T. Head et al, "scikit-optimize/scikit-optimize: v0.5.2," Mar. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1207017

[12] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv:1802.05799 [cs, stat]*, Feb. 2018, arXiv: 1802.05799. [Online]. Available: http://arxiv.org/abs/1802.05799

[13] M. Feurer, J. N. van Rijn, A. Kadra, P. Gijsbers, N. Mallik, S. Ravi, A. Müller, J. Vanschoren, and F. Hutter, "Openml-python: an extensible python api for openml," *arXiv:1911.02490*, 2019.

[14] S. Hettich and S. D. Bay. (1999) The uci kdd archive. [Online]. Available: http://kdd.ics.uci.edu

[15] E. I. Albert Bifet. (2009) Airlines dataset inspired in the regression dataset from elena ikonomovska. the task is to predict whether a given flight will be delayed, given the information of the scheduled departure. [Online]. Available: http://kt.ijs.si/elena_ikonomovska/data.html

[16] I. Guyon, L. Sun-Hosoya, M. Boullé, H. J. Escalante, S. Escalera, Z. Liu, D. Jajetic, B. Ray, M. Saeed, M. Sebag, A. Statnikov, W. Tu, and E. Viegas, "Analysis of the automl challenge series 2015-2018," in *AutoML*, ser. Springer series on Challenges in Machine Learning, 2019. [Online]. Available: https://www.automl.org/wp-content/uploads/2018/09/chapter10-challenge.pdf

[17] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Jan. 2017, arXiv: 1412.6980. [Online]. Available: http://arxiv.org/abs/1412.6980

[18] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," *arXiv:1706.02677 [cs]*, Apr. 2018, arXiv: 1706.02677. [Online]. Available: http://arxiv.org/abs/1706.02677

[19] L. Zimmer, "data_2k.zip," Jan 2020. [Online]. Available: https://figshare.com/articles/dataset/data_2k_zip/11662428/1

[20] "The importance of hyperparameter tuning for scaling deep learning training to multiple GPUs, howpublished = https://aws.amazon.com/blogs/machine-learning/the-importance-of-hyperparameter-tuning-for-scaling-deep-learning-training-to-multiple-gpus/, note = Accessed: 2020-10-08."

[21] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," *arXiv:1611.01578 [cs]*, Nov. 2016, arXiv: 1611.01578. [Online]. Available: http://arxiv.org/abs/1611.01578

[22] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning Transferable Architectures for Scalable Image Recognition," *arXiv:1707.07012 [cs, stat]*, Jul. 2017, arXiv: 1707.07012. [Online]. Available: http://arxiv.org/abs/1707.07012

[23] A. Zela, A. Klein, S. Falkner, and F. Hutter, "Towards Automated Deep Learning: Efficient Joint Neural Architecture and Hyperparameter Search," *arXiv:1807.06906 [cs, stat]*, Jul. 2018, arXiv: 1807.06906. [Online]. Available: http://arxiv.org/abs/1807.06906

[24] A. Zela, J. Siems, and F. Hutter, "NAS-BENCH-1SHOT1: BENCHMARKING AND DISSECTING ONE-SHOT NEURAL ARCHITECTURE SEARCH," p. 20, 2020.

[25] X. Zheng, R. Ji, Q. Wang, Q. Ye, Z. Li, Y. Tian, and Q. Tian, "Rethinking Performance Estimation in Neural Architecture Search," *arXiv:2005.09917 [cs]*, May 2020, arXiv: 2005.09917. [Online]. Available: http://arxiv.org/abs/2005.09917

[26] P. Balaprakash, R. Egele, M. Salim, S. Wild, V. Vishwanath, F. Xia, T. Brettin, and R. Stevens, "Scalable Reinforcement-Learning-Based Neural Architecture Search for Cancer Deep Learning Research," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '19*, pp. 1–33, 2019, arXiv: 1909.00311. [Online]. Available: http://arxiv.org/abs/1909.00311

[27] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural Architecture Search without Training," *arXiv:2006.04647 [cs, stat]*, Jun. 2020, arXiv: 2006.04647. [Online]. Available: http://arxiv.org/abs/2006.04647

[28] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, and F. Hutter, "Auto-sklearn 2.0: The next generation," 2020.

[29] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms," in *Proc. of KDD-2013*, pp. 847–855.

[30] H2O.ai, *H2O AutoML*, June 2017, h2O version 3.30.0.1. [Online]. Available: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html

[31] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore, "Evaluation of a tree-based pipeline optimization tool for automating data science," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: ACM, 2016, pp. 485–492. [Online]. Available: http://doi.acm.org/10.1145/2908812.2908918

[32] P. Gijsbers, E. LeDell, S. Poirier, J. Thomas, B. Bischl, and J. Vanschoren, "An open source automl benchmark," *arXiv preprint arXiv:1907.00909 [cs.LG]*, 2019, accepted at AutoML Workshop at ICML 2019. [Online]. Available: https://arxiv.org/abs/1907.00909